

## MODEL INTEGRATION WITH A TYPED EXECUTABLE MODELING LANGUAGE

Gordon H. Bradley and Robert D. Clemence, Jr.

Naval Postgraduate School, Monterey, California, 93943, USA

### ABSTRACT

Our research [ 2 ], [ 4 ] extends contemporary executable modeling languages for mathematical programming by defining a typing system for all the objects in a model and by specifying formal methods to manipulate the type information. The modeler's intent to formulate consistent, meaningful constraints and functions can be automatically verified. Here we show how typing supports the development of integrated models from distinct model components. We proposed the "library unit" as an extension to modeling languages that provides a mechanism to build integrated models from previously validated models.

### INTRODUCTION

There have been many successful applications of operations research/management science models to specific operational problems (for example, transportation, scheduling, production planning, capital budgeting). But there have been relatively few successes constructing more comprehensive models that integrate several of these specific models. Although the parallel developments in algorithms and computer technology over the past 30 years have greatly expanded our capacity to solve much larger and more complex models, there have not been corresponding advances in model management techniques to integrate models. The integration of distinct models is sometimes called the logical dimension of integration to distinguish it from the executable dimension that is concerned with how to solve the resulting model.

The traditional approach to integrated model construction is for the modeler to study the component models and then to construct a completely new model. In this approach the modeler is responsible for understanding the relationships among the component models but these relationships and the identity of the component models are suppressed to form the integrated model. The resulting model is distinct from the component models and indeed it could have been constructed directly without reference to them. We will

refer to this model representation as "monolithic" because the model stands alone without any reference to the component models. Much research in model management has been directed toward developing alternatives to this approach.

Executable modeling languages have been developed to provide automated support for the construction and validation of models. Our research [ 2 ], [ 4 ] extends contemporary executable modeling languages by specifying a typing system for models that assigns a type to each model object and then verifies if the modeler has formulated consistent meaningful constraints and functions. Here we introduce typing and then show how typing can be used to support the traditional development of monolithic integrated models. We then propose the "library unit" as an extension to executable modeling languages. The library unit allows the specification of an integrated model to be in terms of the component models and then the computer rather than the modeler constructs the monolithic model. Typing supports this construction by allowing automatic verification that the resulting model has meaningful constraints and functions as intended by the modeler.

### EXECUTABLE MODELING LANGUAGES

There is much current research interest in the design and implementation of executable modeling languages (EML), see references [ 1 ] through [ 14 ]. Although much of this work is focused on the definition of mathematical programming models, it involves the definition of operations research/management science models using algebraic notation and thus has potential application to simulation, regression, queuing and other models. The traditional approach to defining mathematical programming models has been to describe the model in an informal algebraic notation and then to develop a matrix generator computer program to construct the problem in the form required by the software system that will optimize the problem. The two phase approach has significant drawbacks as discussed for example in [ 2 ], [ 4 ], [ 8 ].

EMLs offer an alternative to the matrix generator approach by specifying a formal modeling language to define the model and then making the computer, not the modeler, responsible for the construction of the problem in the form that the optimizer software requires. The modeler conceives, records, and validates his model using a formal modeling language with algebraic notation that also documents the model. The modeling language definition of the model is read by the computer, translated to the form required by the optimizer software, optimized, and the solution returned for analysis, all without human intervention. Production [ 13 ] and experimental [ 3 ], [ 9 ], [ 14 ] versions of EMLs exist.

An EML is a declarative language that is formal in the sense that it has an unambiguous syntax and semantics. Any EML must satisfy two potentially conflicting sets of requirements: first, it must be convenient and intuitive for people and as expressive and as easy to use as the informal notation that it replaces and, second, it must be formal and use computer compatible notation so it can be processed by a computer.

#### TYPE CALCULUS FOR EXECUTABLE MODELING LANGUAGES

One aspect of model development that has received little automated support is the verification that the algebraic representation of the model correctly represents the modeler's intention. This intention is expressed in the form of explanatory descriptions which are associated with each numeric-valued symbol in the model. These descriptions are a necessary part of any modeling effort because they assign real world meaning to model data and computation results. The computer manipulates numbers - the meaning assigned to the data and the results is the responsibility of the modeler who is doing a "dimensional" check of each model function and constraint. This is done by replacing each numeric-valued symbol with its explanatory description and then applying two kinds of dimensional calculus. One is the calculus of measurements units: multiplication and division of units and a unit analysis to verify that pure numbers that are added, subtracted or compared have the same scale of reference. The other kind is concerned with what the symbol represents (e.g. apples, cars) and which of its properties are being measured (e.g. cost, height, weight/time). Again the calculus prescribes the rules for multiplication, division, addition, subtraction and comparison. We call the explanatory description of model symbols together with the calculus to manipulate them a typing system.

Our research [ 2 ], [ 4 ] extends contemporary EMLs by specifying a typing system for an extended version of dimensional systems and specifying formal methods to manipulate the type information. For an EML with typing a computer system can automatically

determine if a model is well formed in the sense that functions and constraints do not include typing errors. The computer can verify the modeler's intention to formulate consistent, meaningful constraints and functions. This formalizes and thus allows automation of what in contemporary EMLs are only comments that require human validation.

The modeler assigns each variable, parameter, constant, function and constraint in the model a type that consists of a concept description, quantity description and unit description. The concept represents the essence of the object, for example, APPLES, STEEL, COST, LABOR HOURS. A quantity is an attribute of the concept that is measurable, for example, HEIGHT, CARDINALITY, WEIGHT/LENGTH<sup>2</sup>, HARDNESS. For each quantity there is a standard unit of measurement with optional scale factor. The units are from specified unit systems with conversion factors between units, for example, FEET from English Length, [100]TONS from Avoirdupois Weight, POUNDS/INCH<sup>2</sup> from both. An example of a type is:

LENGTH	of	@CAR	in	FEET
quantity		concept		unit

Concepts are prefixed with the symbol @ to distinguish them from quantities. Concepts are unchanged by multiplication and division while quantities and units are subject to the usual rules. For example, WEIGHT of @BOXES OF APPLES in POUNDS divided by VOLUME of @BOXES OF APPLES in INCH<sup>2</sup> yields WEIGHT/VOLUME of @BOXES OF APPLES in POUNDS/INCH<sup>2</sup>.

For the operations of addition, subtraction, comparison and assignment, both objects must have the same type (that is the same concept, quantity and unit). If the types are not equal the system can automatically do conversions. For example, if one unit is INCH and the other is FEET, the system converts one to the other. If one concept is @APPLES and the other is @ORANGES, a user supplied conversion can convert both to @FRUIT. If the system is unable to convert one or both to make the types equal, an error is indicated.

The unit systems and the conversion factors among their components are built into the system, for example, English Length (INCH, FEET, YARD, MILE). Quantity conversions can be specified by the user, for example, WEIGHT/VOLUME <- -> DENSITY. Concept conversion is specified by one way conversions @FRUIT <- - @APPLES (but not the reverse). The concept conversions are specified in a concept graph that contains all the possible conversions.

All contemporary EMLs allow the definition of sets and provide operations to construct new sets. Variables, parameters, functions and constraints can be

defined over the sets. In some instances, all the objects defined over a set have the same type, but it is also possible to have a unique type for each element. For example, for the variables in a transportation problem  $\text{FLOW}(i, j)$ , the type for  $\text{FLOW}(i, j)$  can be  $\text{@BUTTER}(i, j)$ ; this means that the concept of  $\text{FLOW}(\text{NY}, \text{BOSTON})$  is  $\text{@BUTTER}(\text{NY}, \text{BOSTON})$  and it is different from  $\text{FLOW}(\text{NY}, \text{PHIL})$  with concept  $\text{@BUTTER}(\text{NY}, \text{PHIL})$ .

The type system can be added to any EML. For the examples we have developed a "generic" EML specified in [ 4 ] that contains the features of several existing systems, for example [ 1 ], [ 3 ], [ 9 ], [ 10 ], [ 11 ], [ 13 ]. Figure 1 is a transportation problem expressed in the generic EML. The typing information is contained

within the symbols  $\langle\langle \rangle\rangle$ ; the unit information is further enclosed within  $\# \#$ . In this brief description of typing we rely on the reader's intuition about dimensional systems.

Each numeric-valued object has concepts preceded by @, quantities, and units with optional scale factors. For example, the variables  $X(i, j)$  have concept  $\text{@GOOD}(i, j)$ , quantity  $\text{WEIGHT\_PER\_PERIOD}$  and unit  $\text{LBS/DAY}$  with scale factor 100. Note that  $\text{@GOOD}(i, j)$  is a distinct concept for each  $(i, j)$  pair. Because  $\text{@GOOD}(i, j)$  in the type of  $C(i, j)$  cancels  $\text{@GOOD}(i, j)$  in the type of  $X(i, j)$ , the objective function is type valid and yields  $\text{COST}$  of  $\text{@TRANSPORT}$  in  $\text{US\_\$}$ .

```

<< CONCEPT GRAPH
  @*          <--  @TRANSPORT [ COST ]
  @*          <--  @GOOD(i, .) [WEIGHT_PER_PERIOD]
  @*          <--  @GOOD(., j) [WEIGHT_PER_PERIOD]
  @GOOD(i, .) <--  @GOOD(i, j)
  @GOOD(., j) <--  @GOOD(i, j) >>

<< UNIT SYSTEMS
  WEIGHT_PER_PERIOD : Avoirdupois_Weight/Standard_Time
  COST               : US_Currency >>

SETS
  SOURCE i ; << nominal >>
  SINK j   ; << nominal >>
  ARC(i, j) := {SOURCE} x {SINK} ;

VARIABLES
  X(i, j) {ARC}; << WEIGHT_PER_PERIOD of @GOOD(i, j) # [100] LBS/DAY # >>
  POSITIVE: X(i, j);

PARAMETERS
  S(i) {SOURCE} ; << WEIGHT_PER_PERIOD of @GOOD(i, .) # [100] LBS/DAY # >>
  D(j) {SINK} ; << WEIGHT_PER_PERIOD of @GOOD(., j) # [100] LBS/DAY # >>
  C(i, j) {ARC} ; << COST OF @TRANSPORT / WEIGHT_PER_PERIOD of @GOOD(i, j)
    #US_$ / ( [100] LBS/DAY ) # >>

FUNCTIONS
  OBJ := SUM (i, j) {ARC} ( C(i, j)*X(i, j) ) ; << COST of @TRANSPORT # US_$ # >>

CONSTRAINTS
  SUPPLY(i) {SOURCE} := SUM (j) {ARC} ( X(i, j) ) =E= S(i) ; <<WEIGHT_PER_PERIOD of @GOOD(i, .)
    # [100] LBS/DAY # >>
  DEMAND(j) {SINK} := SUM (i) {ARC} ( X(i, j) ) =L= D(j) ; <<WEIGHT_PER_PERIOD of @GOOD(., j)
    # [100] LBS/DAY # >>

```

Figure 1 Transportation Model in Generic Typed EML

Since  $@GOOD(i, j)$  is unique for each  $(i, j)$ , conversion of concepts is necessary before the summation of  $X(i, j)$  can be allowed in the constraints. It is the modeler's intent that it is valid to sum  $X(i, j)$  that originate at the same source or  $X(i, j)$  that terminate at the same sink and all other combinations are invalid. We introduce the concept  $@GOOD(i, .)$  which is the good that originates at a specific source  $i$  and terminates at any sink and concept  $@GOOD(., j)$  which is the good that originates at any source and terminates at a specific sink  $j$ . Using the concept graph the system automatically performs valid concept conversions. Figure 2 is another representation of the concept graph, the system performs as necessary "upward" conversions of concepts.

The typing system also includes classifying sets and then checking the operations performed on set elements. The designation "nominal" means that the only allowable operations on the set elements are equal, not equal and membership. Sets designated "ordinal" additionally allow ordering operations and "interval" additionally has an integer associated with the element's ordinal position [ 4 ]. Full details on typing including the typing of input data and output reports is included in [ 4 ].

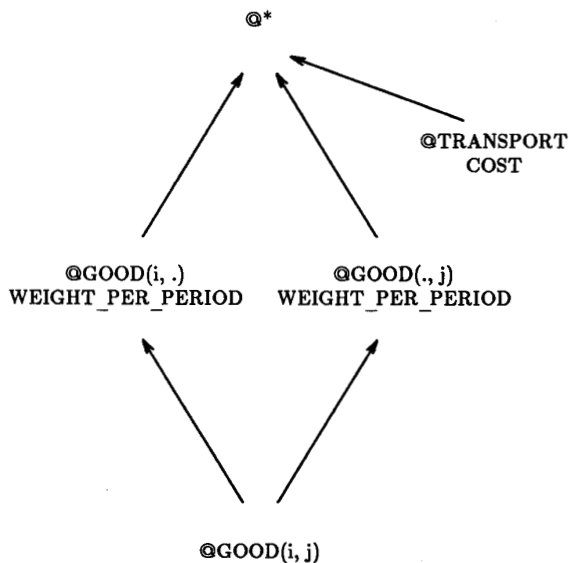


Figure 2 Concept Graph For Transportation Model

## INTEGRATION WITH A TYPED EML

The inclusion of typing in an EML provides additional automatic support for the logical integration of distinct models. This is illustrated in this section by

describing the integration of two models using the capabilities of existing contemporary EMLs. As discussed above, in these systems the modeler has the responsibility to conceive the new model and to then construct a "monolithic" representation using the EML. Since the EML has no special capabilities that support the integration of distinct models, the modeler must construct a representation of the new model that suppresses the relationships of model features to the component models.

The benefits of typing will be illustrated by integrating two transportation models by superimposing the sink nodes in the first model on the source nodes in the second. This can be imagined to be an organization with plants on the West Coast, warehouses in the Midwest and customers in the East. Two distinct models have been developed and validated. The first minimizes the cost of shipping a single good from the plants to the warehouses (with limited warehouse capacity). Using the optimal shipments from the first model at each warehouse as fixed supply, the second model minimizes shipping costs to the customers. This procedure is, in general, suboptimal; an integrated model with the warehouses as capacitated transshipment points will construct a globally optimal solution.

Both models are particular cases of Figure 1. The integration will be described using a text editor as the main support tool. Each model will be modified separately, the models will then be merged and finally additions will be made to the integrated model. For the WEST model, eliminate the  $@* < - @GOOD(., j)$  link in the concept graph. For the EAST model, eliminate the  $S(i)$  parameter, the  $SUPPLY(i)$  constraints, the first 3 links in the concept graph and the units systems. In the EAST model, replace SINK with CUSTOMER and replace  $k$  with  $j$  everywhere. Eliminate the SOURCE set and then replace SOURCE with SINK and replace  $i$  with  $j$  everywhere. Change the names of objects  $X, D, C, ARC, OBJ$  and DEMAND to  $EAST.X, EAST.D, EAST.C, EAST.ARC, EAST.OBJ$  and  $EAST.DEMAND$  respectively. Merge the model files and add a function  $TOTAL\_OBJ$  and constraints  $TRANSSHIP(j)$ . Add concept conversion  $@* < - @GOOD(*, j), @GOOD(*, j) < - @GOOD(., j)$  and  $@GOOD(*, j) < - @GOOD(j, .)$ . The new concept conversions allow  $\sum_i X(i, j)$  to be added to  $\sum_k EAST.GOOD(j, k)$ . Eliminate duplicate statements. The result is Figure 3.

As illustrated, the bulk of support for the integration is accomplished with a text editor not by the modeling language. At the modeler's direction, the text editor concatenates model files, eliminates unwanted language statements, moves blocks of text, and finds/replaces designated character strings. An EML (with or without typing) supports the integration by providing several useful consistency checks. First, the EML enforces the principle that all model objects must

```

<< CONCEPT GRAPH
    @*          <-- @TRANSPORT [ COST ]
    @*          <-- @GOOD(i, .) [WEIGHT_PER_PERIOD]
    @GOOD(i, .) <-- @GOOD(i, j)
    @GOOD(., j) <-- @GOOD(i, j)
    @*          <-- @GOOD(., k) [WEIGHT_PER_PERIOD]
    @GOOD(j, .) <-- @GOOD(j, k)
    @GOOD(., k) <-- @GOOD(j, k)
    @*          <-- @GOOD(*, j)[WEIGHT_PER_PERIOD]
    @GOOD(*, j) <-- @GOOD(., j)
    @GOOD(*, j) <-- @GOOD(j, .) >>

<< UNIT SYSTEMS
    WEIGHT_PER_PERIOD : Avoirdupois_Weight/Standard_Time
    COST               : US_Currency >>

SETS
    SOURCE i          ;    << nominal >>
    SINK j             ;    << nominal >>
    ARC(i, j)         := {SOURCE} x {SINK} ;
    CUSTOMER k        ;    << nominal >>
    EAST.ARC(j, k)    := {SINK} x {CUSTOMER} ;

VARIABLES
    X(i, j) {ARC}; << WEIGHT_PER_PERIOD of @GOOD(i, j) # [100] LBS/DAY # >>
    POSITIVE: X(i, j);
    EAST.X(j, k) {EAST.ARC}; << WEIGHT_PER_PERIOD OF @GOOD(j, k) # [100] LBS/DAY # >>
    POSITIVE: EAST.X(j, k);

PARAMETERS
    S(i) {SOURCE} ; << WEIGHT_PER_PERIOD of @GOOD(i, .) # [100] LBS/DAY # >>
    D(j) {SINK} ; << WEIGHT_PER_PERIOD of @GOOD(., j) # [100] LBS/DAY # >>
    C(i, j) {ARC} ; << COST OF @TRANSPORT / WEIGHT_PER_PERIOD of @GOOD(i, j) #US_$ / ( [100] LBS/DAY ) # >>
    EAST.D(k) {CUSTOMER} ; << WEIGHT_PER_PERIOD of @GOOD(., k) # [100] LBS/DAY # >>
    EAST.C(j, k) {EAST.ARC} ; << COST of @TRANSPORT / WEIGHT_PER_PERIOD of @GOOD(j, k)
    # US_$ / ( [100] LBS/DAY ) # >>

FUNCTIONS
    OBJ := SUM (i, j) {ARC} ( C(i, j)*X(i, j) ) ; << COST of @TRANSPORT # US_$ # >>
    EAST.OBJ := SUM (j, k) {EAST.ARC} (EAST.C(j, k)*EAST.X(j, k)); << COST of @TRANSPORT # US_$ # >>
    TOTAL_OBJ := OBJ + EAST.OBJ ; << COST of @TRANSPORT # US_$ # >>

CONSTRAINTS
    SUPPLY(i) {SOURCE} := SUM (j) {ARC} ( X(i, j) ) =E= S(i) ; <<WEIGHT_PER_PERIOD of @GOOD(i, .)
    # [100] LBS/DAY # >>
    DEMAND(j) {SINK} := SUM (i) {ARC} ( X(i, j) ) =L= D(j) ; <<WEIGHT_PER_PERIOD of @GOOD(., j)
    # [100] LBS/DAY # >>
    EAST.DEMAND(k) {CUSTOMER} := SUM (j) {EAST.ARC} ( EAST.X (j, k) ) =L= EAST.D(k) ;
    << WEIGHT_PER_PERIOD of @GOOD(., k) # [100] LBS/DAY # >>
    TRANSSHIP(j) {SINK} := SUM(i) {ARC} (X(i, j)) =E= SUM(k) {EAST.ARC} (EAST.X(j, k)) ;
    <<WEIGHT_PER_PERIOD of @GOOD(*, j) # [100] LBS/DAY # >>

```

Figure 3 Integrated Transportation Model

be defined before they can be used and identifies any needed objects that have been erroneously eliminated. Second, a cross-reference of the model can identify objects that should be eliminated because they are not used.

An EML with typing contains information about the meaning of the model objects and thus can automatically check some aspects of the integration that would otherwise be the responsibility of the modeler to do by hand. For the above example there are certain necessary validation criteria that can be checked by an EML with typing. First, the integration is valid only if the combined objective functions have identical concept, quantity and units. As mentioned above and described in detail in [ 2 ] and [ 4 ], an EML with typing can do automatic conversions based on the concept graph, quantity conversions and unit conversions. If the types of the two objective functions are not identical or can not be automatically converted so as to be identical, an error is indicated. Second, the replacement of the set of sources in the East model by the set of sinks in the West model is valid only if the source index in the East model is "nominal", that is, no index operations may assume any ordering (e.g. St. Louis < Chicago) or involve arithmetic operations (e.g. St. Louis + 2). The type validation checks all index operations. Third, the type of the decision variables from the two models must be identical (or can be automatically converted to be so) for the transshipment constraint to be valid. The variables must represent the same good measured in the same quantity over the same time interval all in the same units. Typing allows the responsibility of checking these items to be moved from modeler to computer.

## INTEGRATION WITH A MODEL LIBRARY

Although an integrated model may be conceived as the interconnection of several distinct models, contemporary modeling languages require that it have a monolithic representation that suppresses the connections. A model representation must satisfy the language grammar, must contain unique names, must obey the define before use principle and must be expressed in a single textfile. In the previous section these requirements were achieved manually by the modeler using a text editor. The principle disadvantage of this approach is that it requires the modeler to deal with all the variables, parameters, functions, constraints, index sets and types of all the component models simultaneously. The abstraction of the connection of distinct models is lost in the sea of details when forming an integrated model that must have a monolithic representation.

In this section we present an alternate integration mechanism that preserves the identity of each component model and emphasizes model interconnections

while suppressing diversionary detail. This more abstract representation of an integrated model is based on the definition of a "library unit" and on a collection of operations for its manipulation.

The integrated model of the previous section will be represented using a library unit. We will view Figure 1 as a model in a library of validated models. Figure 1 will be referred to as &TRANSPORT and we will first create the WEST model and then the EAST model by separate calls to the library unit and then add new components to form the integrated model. See Figure 4.

"LIB" is a modeling language keyword that causes an inline substitution of an exact or a modified copy of an archival model, called a "library unit". The names of library units are prefaced by the "&" character. Instances of library units are identified by names beginning with the "%" character. The character string "%EAST := &TRANSPORT" indicates that the left argument is an instance of the right argument.

The differences between the instance and the original are detailed after the keyword "WHERE". If the instance is an exact copy, this keyword is omitted. The keyword "END" is used to mark the end of the library unit modifications. Three special operators are employed in this description. Operations are applied sequentially; each one assumes that the operations that precede it have been done. The "<-" operator replaces the character string at the head of the arrow with the character string at the tail. The "<=" operator is a type constrained version of the "<-". It has three actions: it erases the definitions of the typed object on the left of the operator in the instance; it replaces the character string on the left with the character string on the right; and, it inserts an assertion into the text of the model that the type of the right argument is equivalent to the type of the left argument. This assertion is tested during type verification. If the assertion is false, an error message is generated. In this example, the assertion would be

```
<< ASSERTION: type(SINK j) =?= nominal >>
```

The third operator used in the example is "ELIM( )". This operator eliminates the named objects included within its parentheses from the instance. This involves masking object declarations and replacing the names of numerical objects in arithmetic expressions by "0" and "1". The "0" is used when the object is an operand in addition, subtraction or a relational operation. The "1" is used when the object is an operand in multiplication or division.

To preclude ambiguity with two versions of a single library model, the names in the WEST and EAST models need to be distinguished. In the copy of &TRANSPORT instantiated with the %EAST call to the library, the names of its objects are prefaced with

```

LIB    % := &TRANSPORT WHERE
      ELIM (@ * <- - @GOOD(., j));
      END
LIB %EAST := &TRANSPORT WHERE
      ELIM (S(i), SUPPLY (i));
      SINK <- CUSTOMER;
      j <- k;
      SOURCE <= SINK;
      i <- j;
      END
FUNCTIONS
  TOTAL OBJ := OBJ + EAST.OBJ << COST OF @TRANSPORT #US_ $#>>;
CONSTRAINTS
  TRANSSHIP(j) {SINK} := SUM(i) {ARC} (X(i, j) =E= SUM(k) {EAST.ARC} (EAST.X(j, k))
    <<WEIGHT_PER_PERIOD OF @GOOD(*, j) # [100] LBS/DAY # >>
CONCEPT GRAPH
  @* <- - @GOOD(*, j) [WEIGHT_PER_PERIOD]
  @GOOD(*, j) <- - @GOOD(., j)
  @GOOD(*, j) <- - @GOOD(j, .)

```

Figure 4. Integrated Transportation Model Using Library Units

its instance name, EAST, followed by a period. For the first call with just % the names from the library are used without change. The above modeling language statements produce a modeling language textfile identical to Figure 3.

Four other examples of model integration using a typed EML including the integration of a production model with a transportation model are developed in [4].

## LIBRARY UNITS

A library unit is a model or fragment of a model that has been kept as a template for building new, perhaps integrated, models. Each library unit has four parts: a type context, a body, a unique name and an interface. The type context contains a concept graph and a measurement system. For example, in Figure 1 the quantity COST is attributed to the concept @TRANSPORT and measured in US Currency. The body of a library unit is a typed modeling language representation that can contain index sets, parameters, variables, functions and constraints. The body may be a complete model or a model fragment that contains, for example, data transformations or a collection of constraints. Model fragments, however, are still required to satisfy the define before use principle. Each type used in the body is derivable from the concepts, quantities and measurement systems declared in the type context.

The type context and body of a library unit are summarized by a unique name and manipulated through two lists of arguments, called an interface. One list is headed by the relabel operator , "<-", the other

by the replacement operator , "<=". The presence of a label, index suffix, etc. in a list means that it is a legal left-hand argument for the operator that heads the list. While any character string in the type context or body can appear in the relabel list, only names of typed objects (e.g., variables) can appear in the replacement list. Any index set, parameter, variable, function or constraint in the library unit is a legal argument for the "ELIM( )" operator. The contents and organization of the interface are specified by the designer of the library unit to control its usage. When no interface is specified the only permissible use of the library unit is to copy it verbatim. We envision that a call on a library unit using the "LIB" keyword would be embedded in a model as a macro expansion that would yield multiple typed modeling language statements. Before such a model would be submitted to a modeling language translator and type analyzer each "LIB" statement would be replaced by its expanded form and then duplicate statement would be eliminated. The job of expanding library unit references would be performed by a separate preprocessor. The output of the preprocessor would be a typed modeling language textfile. This full form of the model would then be submitted directly to the modeling language translator or, if desired, revised manually by the user before further processing.

In summary, the library unit is intended as means of saving and reusing models. Reuse is facilitated through the provision of special operators for relabeling, for replacing typed objects and for eliminating modeling language objects. These features automate many of the tedious, repetitive symbol manipulations that currently are done to tailor models for new applications.

## INTEGRATION WITH LIBRARY UNITS

One obvious advantage of a library unit or any archival model is that it allows modelers to build upon the work of others. The importance of the library unit construct to integrated modeling is its power as an abstraction and as executable documentation. First, by summarizing a model by a unique name and an interface of arguments, the library unit suppresses diversionary detail and emphasizes the modeling constructs that bind component models together. Second, integrated models constructed by combining modeling language statements and library unit invocations provide an executable record of how the full model submitted for model validation was derived from its components. In addition, the use of "instance\_name" prefixes on modeling language identifiers, preserves the origin of each construct assumed from a component model.

## ACKNOWLEDGEMENTS

The first author would like to acknowledge the support of the Mathematics Group of the Office of Naval Research. The authors would like to thank Art Geoffrion and Dan Dolk for introducing them to executable modeling languages.

## REFERENCES

- [1] Bisschop, J. and Meeraus, A., "On the Development of a General Algebraic Modeling System in a Strategic Planning Environment," Math. Programming Stud. 20 (October 1982), North-Holland, Amsterdam, 1-29.
- [2] Bradley, G. H. and Clemence, Jr., R. D., "A Type Calculus for Executable Modeling Languages", Technical Report NPS52-87-029, Naval Postgraduate School, July 1987. (Presented at the Martin Beale Memorial Symposium, London, England, July 6-8, 1987; to appear in IMA Journal of Mathematics in Management.)
- [3] Clemence, Jr., R. D., "LEXICON: A Structured Modeling System for Optimization," Master's Thesis, Naval Postgraduate School, Monterey, CA, June 1984.
- [4] Clemence, Jr., R. D., "A Type Calculus For Modeling Languages," Ph.D. Dissertation, Naval Postgraduate School, Monterey, CA (in preparation).
- [5] Dolk, D. R., "Data as Models: An Approach to Implementing Model Management," Decision Support Systems, 2, 1 (March 1986), 73-80.
- [6] Dolk, D. R., "A Generalized Model Management system for Mathematical Programming," ACM Trans. Math. Software, 12, 2 (June 1986), 92-125.
- [7] Dolk, D. R. and Konsynski, B., "Model Management in Organizations," Information and Management, 9, 1 (August 1985), 35-47.
- [8] Fourer, R., "Modeling Languages Versus Matrix Generators for Linear Programming," ACM Trans. Math. Software, 9, 2 (June 1983), 143-183.
- [9] Fourer, R., Gay, D. M. and Kernighan, B. W., "AMPL: A Mathematical Programming Language," Computing Science Technical Report No. 133, AT&T Bell Laboratories, Murray Hill, NJ 07974, January, 1987.
- [10] Geoffrion, A. M., Structured Modeling, unpublished manuscript, January 1986.
- [11] Geoffrion, A. M., "An Introduction to Structured Modeling," Management Science, 33, 12 (May 1987), 547-588.
- [12] Geoffrion, A. M., "Integrated Modeling Systems," Working Paper No. 343, Western Management Science Institute, UCLA, November 1986.
- [13] Kendrick, D. A. and Meeraus, A., GAMS: An Introduction, The World Bank, January 1987, The Scientific Press, Palo Alto, to appear.
- [14] Lucas, C. and Mitra, G., "Computer Assisted Mathematical Programming (Modeling System: CAMPS)", (To appear in Computer Journal)